Bits & Bots Database Project

Zhuo Xu

CSE 3241
Autumn 2020
Professor: Pichkar
Grader: Logan Laurer

Table of Contents

Part 1 – The Final Report

Section 1 – Database Description

A. EER Model / EERD

High quality picture of the EERD will go here (in landscape)

Note: in our system an Order can contain only one product. However, a Buyer can still purchase multiple IP items (which are perhaps from different stores) at the same time. The is facilitated through the Shopping Cart entity. A Shopping Cart can contain many different products, which are perhaps from different stores. Then, when the user hits "checkout", what would happen is that one order would be created for every Product in the Buyer's shopping cart. Thus the Buyer can purchase multiple different IP items (which are perhaps from different stores) all in one click.

B. Relational Schema / Database Schema

High quality picture of the Database Schema will go here (in landscape)

C. Functional Dependencies and Normalization

- **User**
  - {UserID} → {Name, Email, Password, Paypal Info, Venmo Info, Karma Points, Cryptocurrency Info}
  - {Email} -> {UserID}
  - {Email} -> {Name}
  - {Email} -> {Password}
  - {Email} -> {Paypal Info}
  - {Email} -> {Venmo Info}
  - {Email} -> {Karma Points}
  - {Email} -> {Cryptocurrency Info}
  - **2NF:** All values are atomic (so 1NF); all non-key attributes are full dependent on the key, but there are some transitive dependencies (eg, UserID key -> Email, and Email -> Name) so not 3NF.
  - **Justification for not bringing this table to 3NF**: The idea with normalization is that it forces you to remove duplicate data in your tables. Normally for tables in 2NF but not 3NF, going to 3NF will allow you to remove data redundancy (i.e., have data defined in only one spot). However, in this case there is a 1-to-1 mapping between UserIDs and Emails (i.e., one UserID is always exactly associated with one Email; and one Email is always exactly associated with one user). Hence, there would be no duplication of data here. So we argue that it would make the design more complicated to split the table up and achieve 3NF. Hence we leave the table as is.
- **Buyer**
  - None
  - **BCNF**: 1NF because all attributes are atomic. 2NF because all non-key attributes depend on the entire primary key. 3NF because there are no transitive dependencies on non-key attributes. BCNF because all determinants are candidate keys.
- **Credit/Debit Card**
  - {CardNumber} → {CVV, Exp Date, Cardholder Name, UserID}
  - **BCNF**: 1NF because all attributes are atomic. 2NF because all non-key attributes depend on the entire primary key. 3NF because there are no transitive dependencies on non-key attributes. BCNF because all determinants are candidate keys.
- **Shopping Cart**
  - None
  - **BCNF**: 1NF because all attributes are atomic. 2NF because all non-key attributes depend on the entire primary key. 3NF because there are no transitive dependencies on non-key attributes. BCNF because all determinants are candidate keys.

- **<u>Shopping Cart Product</u>**
    - None
    - **BCNF**: 1NF because all attributes are atomic. 2NF because all non-key attributes depend on the entire primary key. 3NF because there are no transitive dependencies on non-key attributes. BCNF because all determinants are candidate keys.
- **<u>Seller</u>**
    - {SellerID} ⟶ {Bio}
    - **BCNF**: 1NF because all attributes are atomic. 2NF because all non-key attributes depend on the entire primary key. 3NF because there are no transitive dependencies on non-key attributes. BCNF because all determinants are candidate keys.
- **<u>Seller Photos</u>**
    - {SellerPhotoID} ⟶ {SellerId, Photo}
    - **BCNF**: 1NF because all attributes are atomic. 2NF because all non-key attributes depend on the entire primary key. 3NF because there are no transitive dependencies on non-key attributes. BCNF because all determinants are candidate keys.
- **<u>Store</u>**
    - {StoreID} ⟶ {SellerID, Name, Description, Banner}
    - **BCNF**: 1NF because all attributes are atomic. 2NF because all non-key attributes depend on the entire primary key. 3NF because there are no transitive dependencies on non-key attributes. BCNF because all determinants are candidate keys.
- **<u>Store Accepted Payment Types</u>**
    - None
    - **BCNF**: 1NF because all attributes are atomic. 2NF because all non-key attributes depend on the entire primary key. 3NF because there are no transitive dependencies on non-key attributes. BCNF because all determinants are candidate keys.
- **<u>Seller Review</u>**
    - {ReviewID} ⟶ {BuyerID, SellerID, Review}
    - **BCNF**: 1NF because all attributes are atomic. 2NF because all non-key attributes depend on the entire primary key. 3NF because there are no transitive dependencies on non-key attributes. BCNF because all determinants are candidate keys.
- **<u>Store URL</u>**
    - None
    - **BCNF**: 1NF because all attributes are atomic. 2NF because all non-key attributes depend on the entire primary key. 3NF because there are no transitive dependencies on non-key attributes. BCNF because all determinants are candidate keys.
- **<u>Product</u>**
    - {ProductID} ⟶ {StoreID, Price, Title, Description, Availability, File Type, Download Link, Size}
    - **BCNF**: 1NF because all attributes are atomic. 2NF because all non-key attributes depend on the entire primary key. 3NF because there are no transitive dependencies on non-key attributes. BCNF because all determinants are candidate keys.

- **Product Image**
    - {ProductImageID} → {ProductID, Image}
    - **BCNF**: 1NF because all attributes are atomic. 2NF because all non-key attributes depend on the entire primary key. 3NF because there are no transitive dependencies on non-key attributes. BCNF because all determinants are candidate keys.
- **Product Keyword**
    - {ProductKeywordID} → {ProductID, Keyword}
    - **BCNF**: 1NF because all attributes are atomic. 2NF because all non-key attributes depend on the entire primary key. 3NF because there are no transitive dependencies on non-key attributes. BCNF because all determinants are candidate keys.
- **Product Review**
    - {ReviewID} → {ProductID, UserID, Review}
    - **BCNF**: 1NF because all attributes are atomic. 2NF because all non-key attributes depend on the entire primary key. 3NF because there are no transitive dependencies on non-key attributes. BCNF because all determinants are candidate keys.
- **AnOrder**
    - {OrderID} → {UserID, ProductID, DeliveryEmail, AmountPaid, Date, PaymentType}
    - **BCNF**: 1NF because all attributes are atomic. 2NF because all non-key attributes depend on the entire primary key. 3NF because there are no transitive dependencies on non-key attributes. BCNF because all determinants are candidate keys.
- **Message**
    - {MessageID} → {Sender, Receiver, Subject, Text, Attachment, Read, SentTime}
    - **BCNF**: 1NF because all attributes are atomic. 2NF because all non-key attributes depend on the entire primary key. 3NF because there are no transitive dependencies on non-key attributes. BCNF because all determinants are candidate keys.

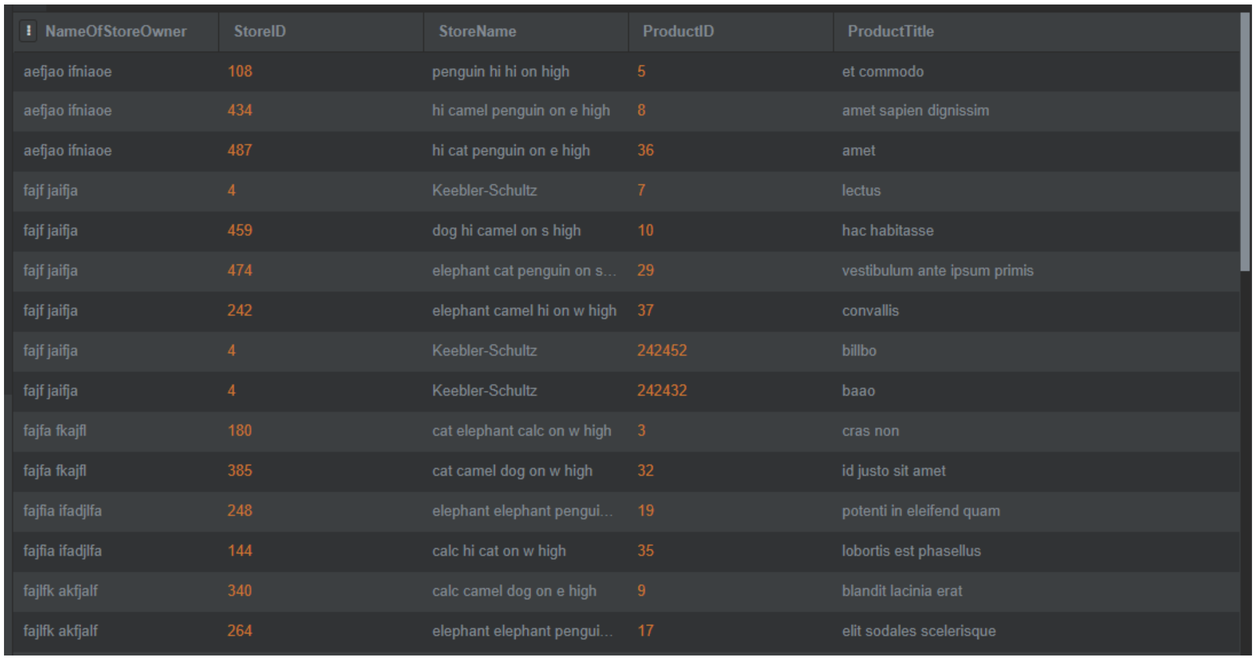
D. Views

View 1: Shows all the Products for all the Stores, and the owner of each Store

This view does as it says -- it shows all the products, for all the stores, and it shows the name of the owner for each store. The rationale for having this view is that it allows someone to see all products, the store it came from, and the owner of that store all in one place.

SQL Code to create this View:

```
CREATE VIEW ProductsInStores AS
        SELECT User.Name AS NameOfStoreOwner, Store.StoreID, Store.Name AS
        StoreName, ProductID, Title AS ProductTitle
        FROM Store, Product, Seller, User
        WHERE Store.StoreID = Product.StoreID AND Store.SellerID=Seller.SellerID AND
        Seller.SellerID=User.UserId;
```



NameOfStoreOwner	StoreID	StoreName	ProductID	ProductTitle
aefjao ifniaoe	108	penguin hi hi on high	5	et commodo
aefjao ifniaoe	434	hi camel penguin on e high	8	amet sapien dignissim
aefjao ifniaoe	487	hi cat penguin on e high	36	amet
fajf jaifja	4	Keebler-Schultz	7	lectus
fajf jaifja	459	dog hi camel on s high	10	hac habitasse
fajf jaifja	474	elephant cat penguin on s...	29	vestibulum ante ipsum primis
fajf jaifja	242	elephant camel hi on w high	37	convallis
fajf jaifja	4	Keebler-Schultz	242452	billbo
fajf jaifja	4	Keebler-Schultz	242432	baao
fajfa fkajfl	180	cat elephant calc on w high	3	cras non
fajfa fkajfl	385	cat camel dog on w high	32	id justo sit amet
fajfia ifadjlfa	248	elephant elephant pengui...	19	potenti in eleifend quam
fajfia ifadjlfa	144	calc hi cat on w high	35	lobortis est phasellus
fajlfk akfjalf	340	calc camel dog on e high	9	blandit lacinia erat
fajlfk akfjalf	264	elephant elephant pengui...	17	elit sodales scelerisque

Relational Algebra Expression that will produce this View:

StoreWithColumnNameChanged $\leftarrow \rho_{\text{(SellerUserId, StoreId, StoreName, StoreDescription, Banner)}}$ (Store)

ProductStoreSellerUser $\leftarrow$ (((Product $\bowtie_{\text{ProductId = StoreId}}$ StoreWithColumnNameChanged) $\bowtie_{\text{SellerId = SellerId}}$ Seller) $\bowtie_{\text{SellerId = UserId}}$ User)

Temp $\leftarrow \pi_{\text{Name, StoreId, StoreName, ProductId, Title}}$ (ProductStoreSellerUser)

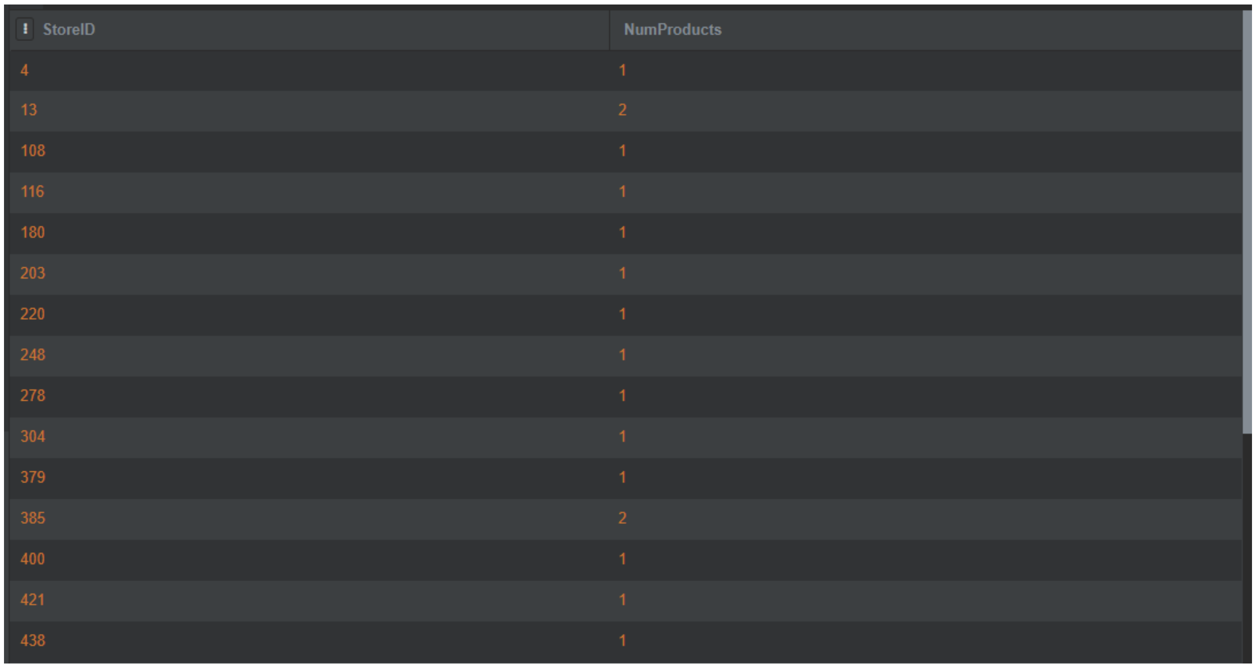
FinalResult $\leftarrow \rho_{\text{(NameOfStoreOwner, StoreId, StoreName, ProductId, ProductTitle)}}$ (Temp)

View 2: Shows number of products purchased from each store

This query first joins Stores and Products based on StoreID and then joins this with Orders based ProductID. From this joined table, the number of products corresponding to each seller is calculated and returned as a view. The rationale for having this query is that this is by far the easiest way to see how many products are sold by each store.

SQL Code to create this View:

```
CREATE VIEW ProductNum (StoreID, NumProducts) AS
        SELECT S.StoreID, count(*)
        FROM Store AS S, Product AS P, AnOrder AS O
        WHERE S.StoreID = P.StoreID AND P.ProductID = O.ProductID
        GROUP BY S.StoreID;
```



StoreID	NumProducts
4	1
13	2
108	1
116	1
180	1
203	1
220	1
248	1
278	1
304	1
379	1
385	2
400	1
421	1
438	1

Relational Algebra Expression that will produce this View:

$$T1 \leftarrow ((\text{AnOrder} \bowtie_{\text{ProductId = ProductId}} \text{Product}) \bowtie_{\text{StoreId = StoreId}} \text{Store})$$

$$T2 \leftarrow {}_{\text{StoreID}}\mathbf{F}_{\text{COUNT OrderId}}(T1)$$

$$\text{FinalResult} \leftarrow \rho_{(\text{StoreId, NumProducts})}(T2)$$

E. Indexes

Index 1: An index on the "File Type" column in the Product table

The idea here is to make lookups based on the file type of a product / IP Item fast, since buyers would very likely want to search for products based on their file type.

For example, maybe a buyer wants to find all products where file type = "pptx". This would correspond to a SQL query like "SELECT * FROM PRODUCTS WHERE FileType="pptx"". Thus, this search would benefit from an index on the FileType column.

Moreover -- maybe buyers want to narrow down their searches based on file type. For instance, say they have searched for all products where Title="Accounting" and FileType="pptx", because they are looking for PowerPoint presentations about Accounting. Here too an index on the File Type column would help; because after all products with the Title=Accounting are found (say via a full table scan), we can use the index to quickly find all products where FileType=pptx. Then, those results can simply be intersected to find all products where Title="Accounting" and FileType=pptx.

Thus, having an index on the FileType column would not only help for searches that are just based on the FileType, but it would also help when users want to narrow down results based on FileType.

All of the queries above are exact-value searches; i.e., we are searching for tuples/rows where we have a one exact value in the FileType column. Or in other words, we are making a search based on a simple equality test (i.e., WHERE FileType="pptx"). Thus, we would use a **Hash-index** for this situation, because Hash indexes are best for exact-value searches.

SQL Code to create this Index:

```
CREATE INDEX FileTypeIndex ON Product(FileType);
```

Note: SQLite documentation states SQLite only supports tree-based index.

Index 2: An index on the "Price" column in the Products table

The idea here is to make lookups based on the Price of a product fast, since users would probably want to do that frequently.

Specifically, we would make this a **B-tree index**, because it's likely that people will lookup products based on price *ranges*, and B-trees are best for queries involving ranges.

For instance, an end-user may want to lookup all products where price < $10.00. If we had this B-tree index on the Price column, that would make this lookup faster. Hence, we think a B-tree index would be a good idea.

We figure it would be more common for an end-user to ask "give me all products where price < 10.00" (a range) then "give me all products where price exactly = 10.00" (an exact-value query). Hence, we would use a B-tree index over a Hash-table index here. Because B-tree indexes are best for range queries, whilst Hash-table indexes are best for exact-value queries.

SQL Code to create this Index:

```
                              CREATE INDEX PriceIndex ON Product(Price);
```

F. Transactions

Transaction 1

This transaction adds 100 karma points to any user that is a seller. This could be useful if there was a time when the managers B&B wanted to give users who owned a store a discount. This would be done by giving sellers karma points for free. We make it a transaction to handle giving all the 100 karma points to the Sellers "at once", so that no Seller appears to get them later than another. This transaction performs one read and one write operation: A read on Seller and a write on User.

SQL Code to run this Transaction:

```
BEGIN TRANSACTION GiveSellersKarmaPoints
        UPDATE USER SET Karma = Karma + 100 WHERE UserID IN (SELECT SellerID
        FROM SELLER);
        IF error THEN GO TO UNDO; END IF;
        COMMIT;
        GO TO FINISH;
        UNDO: ROLLBACK;
        FINISH:
END TRANSACTION;
```

Transaction 2

This transaction retrieves all the items that are currently in a shopping cart. The purpose of this transaction is that it is efficient since it only looks at shopping carts for users that are currently buyers. Also, if a Buyer is in the process of adding items to their shopping cart, the concurrency issues surrounding that are handled by the transaction. It performs 2 read operations, one on ShoppingCartProduct and one on Buyer.

SQL Code to run this Transaction:

```
BEGIN TRANSACTION GetAllShoppingCartItems
        SELECT ProductID FROM ShoppingCartProduct WHERE ShoppingCartID IN (SELECT
        BuyerID FROM Buyer);
        IF error THEN GO TO UNDO; END IF;
        COMMIT;
        GO TO FINISH;
        UNDO: ROLLBACK;
        FINISH:
END TRANSACTION;
```

Section 2 – User Manual

A. Database Description

- **<u>User</u>**
    - User is a generalization entity of Buyer and Seller. Essentially, User simply represents a person who uses Bits & Bots.
    - UserId is an integer and is a primary key used to uniquely identify a user.
    - Name is a string that represents the name of the user.
    - Email is a string for the users email.
    - Password is a string for the user’s password for their account.
    - Paypal is a string representing the username of the user’s Paypal account.
    - Venmo is a string representing the username of the user’s Venmo account.
    - Karma is an integer representing the number of Karma points that the user has.
    - Cryptocurrency is a string representing the user’s cryptocurrency username.
- **<u>Buyer</u>**
    - BuyerId is an integer foreign key from User used to identify a buyer.
- **<u>Credit/Debit Card</u>**
    - UserId is an integer foreign key from User.
    - CardNumber is an integer representing the credit/debit card number for the user and is the primary key.
    - CVV is an integer representing the CVV security code of the user’s credit/debit card.
    - CardholderName represents the name on the user’s card.
    - ExpDate is a date representing the expiration date of the card.
- **<u>Shopping Cart</u>**
    - ShoppingCartId is an integer foriegn key from User identifying the user that the shopping cart corresponds to.
- **<u>Shopping Cart Product</u>**
    - ShoppingCartId is an integer foreign key from ShoppingCart identifying the shopping cart that the shopping cart product corresponds to.
    - ProductID is an integer foreign key identifying the product in the cart.
    - ShoppingCartId and ProductId form the primary key.
- **<u>Seller</u>**
    - BuyerId is an integer foreign key from User used to identify a buyer.
    - Bio is a string containing the bio information for a seller.
- **<u>Seller Photos</u>**
    - SellerPhotoId is an integer primary key used to uniquely identify a Seller Photo.
    - SellerId is an integer foreign key pointing back to the Seller that has this photo.
    - Photo is a string that is the path of the photo on the server.
- **<u>Store</u>**
    - SellerId is an integer foreign key from Seller identifying the owner of the store.
    - StoreId is an integer primary key identifying the store.
    - Name is a string representing the name of the store.
    - Description is a string containing a description/bio for the store.
    - Banner is a string containing a link to a banner photo for the store.

- **Store Accepted Payment Types**
    - StoreId is an integer foreign key from Store identifying the store.
    - Payment is a string representing a payment type (Venmo, Credit/Debit, Paypal, Cryptocurrency).
    - StoreId and Payment together are the primary key.
- **Seller Review**
    - BuyerId is an integer foreign key identifying the buyer who wrote the review.
    - SellerId is an integer foreign key identifying the seller who the review was for.
    - ReviewId is an integer primary key identifying the review.
    - Review is a string containing the actual review.
- **Store URL**
    - StoreId is an integer foreign key identifying the store.
    - URL is a string containing the url for the store.
    - StoreId and URL together form the primary key.
- **Product**
    - StoreId is an integer foreign key from Store identifying the store the product is sold at.
    - ProductId is an integer primary key identifying the product.
    - Price is a decimal denoting the price of the product.
    - Title is a string representing the name of the product.
    - Description is a string containing a description of the product.
    - Availability is a boolean denoting whether or not the product is still in stock.
    - Filetype is a string containing the file extension for the file.
    - Link is a string containing the download link for the product.
    - Size is an integer containing the size of the file.
- **Product Image**
    - ProductId is an integer foreign key from Product identifying the product that the image is of.
    - ProductImageId is an integer foreign key identifying the image.
    - Image is a string containing the link of the image.
- **Product Keyword**
    - ProductId is an integer foreign key from Product identifying the product that the image is of.
    - KeywordId is an integer foreign key identifying the keyword.
    - Keyword is a string containing the keyword.
- **Product Review**
    - ProductId is an integer foreign key from Product identifying the product being reviewed.
    - UserId is an integer foreign key from User identifying the user leaving the review.
    - ReviewId is an integer primary key identifying the review.
    - Review is a string containing the actual review.

- **AnOrder**
    - UserId is an integer foreign key from User identifying the user placing the order.
    - ProductId is an integer foreign key from Product identifying the product being purchased.
    - OrderId is an integer primary key identifying the user.
    - Email is a string identifying the email the product's download link should be sent to.
    - AmountPaid is a decimal denoting the amount paid for the product.
    - Date is a date containing the date the order was placed.
    - PaymentType is a string containing the type of payment that was used (Venmo, Credit/Debit, Paypal, Cryptocurrency).
- **Message**
    - Sender is an integer foreign key from User identifying the sender of the message.
    - Receiver is an integer foreign key from User identifying the receiver of the message.
    - MessageId is an integer primary key identifying the message.
    - Subject is a string containing the subject of the message.
    - Text is a string containing the actual message.
    - Attachment is a string containing a link to an attachment if there is one.
    - WasRead is a boolean denoting whether or not the message has been read yet.
    - Time is a DateTime denoting when the message was sent.

B. Sample Queries to run against our Database

IN SQL (from CP03):

SQL Code	**Meaning of the Query**
From SimpleQueries.txt …...	

SELECT Product.Title
FROM Product, Store, Seller, User
WHERE Product.Price<10 AND Product.StoreID=Store.StoreID AND Store.SellerID = Seller.SellerID AND Seller.SellerID=User.UserID AND User.Name='fajf jaifja';
	Find the titles of all IP Items by Seller with name "fajf jaifja" that cost less than $10

SELECT Product.Title AS ProductPurchased,
AnOrder.Date
FROM AnOrder, Product, User
WHERE AnOrder.ProductID=Product.ProductID AND
AnOrder.UserID=User.UserID AND User.Name='fajf
jaifja';
	Give all names of items purchased and their dates of purchase made by a buyer with name 'fajf jaifja'

SELECT User.Name as SellerName
FROM Product, Store, Seller, User
WHERE Product.Available='TRUE' AND
Product.StoreID=Store.StoreID AND
Store.SellerID=Seller.SellerID AND
Seller.SellerID=User.UserID
GROUP BY SellerName
HAVING Count(SellerName) < 5;
	Find the seller names for all sellers with less than 5 IP Items for sale

SELECT BuyerUser.Name AS NameOfBuyer,
Product.title AS ProductPurchased
FROM AnOrder, Product, Store, Seller, User AS
BuyerUser, User AS SellerUser
WHERE AnOrder.UserID=BuyerUser.UserID AND
AnOrder.ProductID=Product.ProductID AND
Product.StoreID=Store.StoreID AND
Store.SellerID=Seller.SellerID AND
Seller.SellerID=SellerUser.UserID AND
SellerUser.Name='falf iejfla';
	Give all the buyers who purchased a IP Item by a seller with name 'falf iejfla' and the names of the IP Items they purchased

SELECT Count(AnOrder.UserId) AS
NumItemsPurchased
FROM AnOrder, User
WHERE AnOrder.UserID=User.UserID AND
User.Name='fajefai jfiajfl';
	Find the total number of IP Items purchased by a buyer with name 'fajefai jfiajfl'

SELECT BuyerName, MAX(CT) AS
NumItemsPurchased
FROM
(SELECT BuyerUser.Name AS BuyerName,
Count(AnOrder.UserID) as CT
FROM AnOrder, User as BuyerUser
WHERE AnOrder.UserID = BuyerUser.UserID
GROUP BY AnOrder.UserID);
	Find the buyer who has purchased the most IP Items and the total number of IP Items they have purchased

From ExtraQueries.txt	

SELECT U.UserID, S.StoreID
FROM User AS U LEFT OUTER JOIN Store AS S
ON U.UserID=S.SellerID;
	Find the UserId and StoreId for all users, including users who don't have a store

SELECT sum(P.price) as
TotalPriceOfItemsInShoppingCart
FROM User AS U, ShoppingCart AS SC,
ShoppingCartProduct AS SCP, Product AS P
WHERE U.UserID=3 AND
U.UserID=SC.ShoppingCartID AND
SC.ShoppingCartID=SCP.ShoppingCartID AND
SCP.ProductID=P.ProductID;
	Find the total price of items in the shopping cart for the User with UserId=3

SELECT count(*) as NumItemsInShoppingCart
FROM User AS U, ShoppingCart AS SC,
ShoppingCartProduct AS SCP, Product AS P
WHERE U.UserID=3 AND
U.UserID=SC.ShoppingCartID AND
SC.ShoppingCartID=SCP.ShoppingCartID AND
SCP.ProductID=P.ProductID;
	Find the number of Items in the shopping cart for the User with UserId=3

From AdvancedQueries.txt…...	

SELECT Name, sum(AmountPaid) AS TotalSpent
FROM Buyer AS B, User AS U, AnOrder AS O
WHERE BuyerID=U.UserID AND O.UserID=BuyerID
GROUP BY BuyerID;
	Give a list of buyer names, along with the total dollar amount each buyer has spent.

SELECT Name, U.Email
FROM Buyer AS B, User AS U, AnOrder AS O
WHERE BuyerID=U.UserID AND O.UserID=BuyerID
GROUP BY BuyerID
HAVING sum(amountPaid) >
(SELECT avg(TotalSpent)
FROM
(SELECT Name, sum(AmountPaid) AS TotalSpent
FROM Buyer AS B, User AS U, AnOrder AS O
WHERE BuyerID=U.UserID AND O.UserID=BuyerID
GROUP BY BuyerID));
	Give a list of buyer names and e-mail addresses for buyers who have spent more than the average buyer.

SELECT ProductID, count(*)
FROM AnOrder
GROUP BY ProductID
ORDER BY count(*) DESC;
	Give a list of the IP Item names and associated total copies sold to all buyers, sorted from the IP Item that has sold the most individual copies to the IP Item that has sold the least.

SELECT Product.ProductID, Title, SUM(Product.Price)
as TotalDollarProfitsFromThisProduct
FROM Product, AnOrder, User
WHERE AnOrder.ProductId=Product.ProductId AND
AnOrder.UserID=User.UserID
GROUP BY Product.ProductID
ORDER BY TotalDollarProfitsFromThisProduct DESC;
	Give a list of the IP Item names and associated dollar totals for copies sold to all buyers, sorted from the IP Item that has sold the highest dollar amount to the IP Item that has sold the smallest.

CREATE VIEW Temp (SellerID, ProductNum) AS
SELECT Seller.SellerID, count(Seller.SellerID)
FROM ((Seller JOIN Store ON Seller.SellerID = Store.SellerID) JOIN Product ON Store.StoreID = Product.StoreID)
GROUP BY Seller.SellerID;
SELECT SellerID
FROM Temp
WHERE ProductNum = (SELECT max(ProductNum) from TEMP);
	Find the most popular Seller (i.e. the one who has sold the most IP Items)

Select MAX(TotalMadeByThisSeller), IdOfSeller, NameOfSeller
FROM
(SELECT SUM(Product.Price) As TotalMadeByThisSeller, User.UserID as IdOfSeller, User.Name AS NameOfSeller
FROM AnOrder, Product, Store, Seller, User
WHERE AnOrder.ProductId=Product.ProductID AND Product.StoreID=Store.StoreId AND Store.SellerId=Seller.SellerId AND Seller.SellerID=User.UserID
GROUP BY NameOfSeller);
	Find the most profitable seller (i.e. the one who has brought in the most money)

SELECT DISTINCT(U.Name)
FROM User AS U, Buyer AS B, AnOrder AS O, Product AS P, Store AS S, Seller
WHERE B.BuyerID=U.UserID AND O.UserID=B.BuyerID AND O.ProductID=P.ProductID AND P.StoreID=S.StoreID AND Seller.SellerID=
(SELECT IdOfSeller
FROM
(Select MAX(TotalMadeByThisSeller), IdOfSeller, NameOfSeller
FROM (SELECT SUM(Product.Price) As TotalMadeByThisSeller, User.UserID as IdOfSeller, User.Name AS NameOfSeller
FROM AnOrder, Product, Store, Seller, User
WHERE AnOrder.ProductId=Product.ProductID AND Product.StoreID=Store.StoreId AND Store.SellerId=Seller.SellerId AND Seller.SellerID=User.UserID
GROUP BY NameOfSeller)));
	Give a list of buyer names for buyers who purchased anything listed by the most profitable Seller.

CREATE VIEW AmountSpent (BuyerID, Amount) AS SELECT BuyerID, sum(AmountPaid) FROM (Buyer JOIN AnOrder ON BuyerID = UserID) GROUP BY BuyerID;
CREATE VIEW BigBuyers (BuyerID) AS SELECT BuyerID FROM AmountSpent WHERE Amount > (SELECT avg(Amount) FROM AmountSpent);
CREATE VIEW Purchases (ProductID) AS SELECT distinct Product.ProductID FROM ((BigBuyers JOIN AnOrder ON BigBuyers.BuyerID = AnOrder.UserID) JOIN Product ON AnOrder.ProductID = Product.ProductID);
SELECT distinct(Seller.SellerID)
FROM (((Purchases JOIN Product ON Purchases.ProductID = Product.ProductID) JOIN Store ON Product.StoreID = Store.StoreID) JOIN Seller ON Store.SellerID = Seller.SellerID);
	Give the list of sellers who listed the IP Items purchased by the buyers who have spent more than the average buyer.

In Relational Algebra (from CP02):

Relational Algebra Expression	**Meaning of the Query**
Titles $\leftarrow \Pi_{\text{Title}} (\sigma_{\text{Price} < 10}(\sigma_{\text{Seller\_ID = Seller\_ID\_To\_Select}}((\text{SELLER} \bowtie_{\text{Seller\_ID = Seller\_ID}} \text{STORE}) \bowtie_{\text{Store\_ID = Store\_ID}} \text{PRODUCT})))$	Find the titles of all IP Items by a given Seller that cost less than $10
OrderInfo $\leftarrow \Pi_{\text{Title, Date}}(\sigma_{\text{Buyer\_ID = Buyer\_ID\_To\_Select}} ((\text{BUYER} \bowtie_{\text{Buyer\_ID = User\_ID}} \text{ANORDER}) \bowtie_{\text{Product\_ID = Product\_ID}} \text{PRODUCT}))$	Give all the titles and their dates of purchase made by given buyer
Names $\leftarrow \Pi_{\text{Name}} (\text{USER} \bowtie_{\text{User\_ID = Seller\_ID}} (\sigma_{\text{COUNT Product\_ID} < 5} ({}_{\text{Seller\_ID}} F_{\text{COUNT Product\_ID}} ((\text{SELLER} \bowtie_{\text{Seller\_ID = Seller\_ID}} \text{STORE}) \bowtie_{\text{Store\_ID = Store\_ID}} \text{PRODUCT}))))$	Find the seller names for all sellers with less than 5 IP Items for sale
BuyersAndItemNames $\leftarrow \Pi_{\text{Buyer ID, Product Title}}(\text{BUYER} \bowtie_{\text{Buyer\_ID=Buyer\_ID}} (\text{ORDER} \bowtie_{\text{Product\_ID = Product\_ID}} (\sigma_{\text{Seller\_ID = Seller\_ID\_To\_Select}} ((\text{SELLER} \bowtie_{\text{Seller\_ID = Seller\_ID}} \text{STORE}) \bowtie_{\text{Store\_ID=Store\_ID}} \text{PRODUCT})))$	Give all the buyers who purchased a IP Item by a given seller and the names of the IP Items they purchased
BuyerOrders $\leftarrow$ ANOrder $\bowtie_{\text{User\_ID=Buyer\_ID}}$ Buyer OrdersForOurSpecificBuyer $\leftarrow \sigma_{\text{Buyer\_ID=Buyer\_ID\_To\_Select}}$ BuyerOrders Result $\leftarrow F_{\text{COUNT Order\_ID}}$(OrdersForOurSpecificBuyer)	Find the total number of IP Items purchased by a single buyer

BuyerOrders ⟵ AnOrder $\bowtie_{\text{User\_ID=Buyer\_ID}}$ Buyer OrderCountsForEachBuyer ⟵ $_{\text{Buyer\_ID}}F_{\text{COUNT Order\_ID}}$(BuyerOrders) MaxNumberOfItemsPurchased ⟵ $F_{\text{MAX COUNT\_Order\_ID}}$(OrderCountsForEachBuyer) BuyerWithMaxItemsPurchased ⟵ $\sigma_{\text{COUNT\_Order\_ID = MaxNumberOfItemsPurchased.MAX}}$ (OrdersCountsForEachBuyer)	Find the buyer who has purchased the most IP Items and the total number of IP Items they have purchased
DesiredUserCart ⟵ USER $\bowtie_{\text{User\_ID = Desired\_User\_ID}}$ SHOPPING CART ItemsInCart ⟵ (DesiredShoppingCart $\bowtie_{\text{Shopping\_Cart\_ID = Shopping\_Cart\_ID}}$ SHOPPING CART PRODUCT) $\bowtie_{\text{Product\_ID = Product\_ID}}$ PRODUCT CartPrice ⟵ F $_{\text{SUM Price}}$ (ItemsInCart)	Find the price of items in a specific users shopping cart
DesiredUserCart ⟵ USER $\bowtie_{\text{User\_ID = Desired\_User\_ID}}$ SHOPPING CART ItemsInCart ⟵ (DesiredShoppingCart $\bowtie_{\text{Shopping\_Cart\_ID = Shopping\_Cart\_ID}}$ SHOPPING CART PRODUCT) $\bowtie_{\text{Product\_ID = Product\_ID}}$ PRODUCT CartSize ⟵ F $_{\text{COUNT Product\_ID}}$ (ItemsInCart)	Find the number of Items in a specific users shopping cart
UserStore ⟵ $\Pi_{\text{User\_Id, Seller\_user\_id}}$ (USER ⟕ $_{\text{User\_Id = Seller\_user\_id}}$ STORE)	Find the UserId and StoreId for all users, including users who don't have a store

C. INSERT examples

There are likely 4 new entities that you may want to insert into the database: new **buyers**, new **sellers**, new **products**, and new **orders.** We discuss each of them in turn and provide the relevant SQL code.

To add a new Buyer to the database:
In our system, a Buyer is just a special kind of User. Thus, to insert a new Buyer, we start by inserting a new User:

INSERT INTO User(UserID, Name, Email, Password, Paypal, Venmo, Karma, Cryptocurrency) VALUES (1245,'Jared', 'harp.62@osu.edu' ,'hunter2','jahPaypal','jahVenmo',0,'jahBitcoin');

Note that since the UserID is the Primary Key of the User table, the UserId needs to be unique.
Also note that we set the user's Karma to 0 when they start out, since they have not yet earned any karma points.

Next, create a row in the Buyer table to identify this User as a Buyer:

INSERT INTO Buyer(BuyerID) VALUES (1245)

Note that the value for the BuyerID is the same as the one we used for UserID above(=1245). This is required; because BuyerID is a Foreign Key in the Buyer table, it *must* point back to a PK that actually exists in the User table. Thus we need to take care to make BuyerId = the value we used for UserId above.

Having done this, you have created a new Buyer. Do take care to run those steps in order, because if you try to insert the row into the Buyer table first, you will get a "Foreign Key Constraint failed" error because your Foreign Key does not point back to an existing Primary Key in the User table.

To add a new Seller to the database:
In our system, a Seller is just a special kind of User. Thus, to insert a new Seller, we start by inserting a new User:

INSERT INTO User(UserID, Name, Email, Password, Paypal, Venmo, Karma, Cryptocurrency) VALUES (1249,'Sue', 'ellen.2@osu.edu' ,'ehunter2','elPaypal','elVenmo',0,'elBitcoin');

Note that since the UserID is the Primary Key of the User table, the UserId needs to be unique.
Also note that we set the user's Karma to 0 when they start out, since they have not yet earned any karma points.

Next, create a row in the Seller table to identify this User as a Seller:

INSERT INTO Seller(SellerID, Bio) VALUES (12459, 'Check out my stores!')

Note that the value for the SellerID is the same as the one we used for SellerID above(=1249). This is required; because SellerID is a Foreign Key in the Seller table, it *must* point back to a PK that actually exists in the User table. Thus we need to take care to make SellerId = the value we used for UserId above.

Also note that this is when we can supply a Bio for the Seller. The bio is placed in the Seller table, since it is assumed that a Buyer will typically not want to have a bio, thus we did not want to place the Bio in the User table and force all users to have a Bio. (A User can be a Buyer, a Seller, or both in our database).

Having done these two steps, you have created a new Seller.

Do take care to run those steps in order, because if you try to insert the row into the Seller table first, you will get a "Foreign Key Constraint failed" error because your Foreign Key does not point back to an existing Primary Key in the User table.

To add a new Product to the database:
Adding a new Product is as simple as inserting a new row into the Product table:

INSERT INTO Product (StoreID, ProductID, Price, Title, Description, Available, FileType, Link, Size) VALUES (119, 101342,'2.35','virus1','"it is literally a virus"','TRUE','exe','https://fardork.com', 3242);

Do note these two requirements though:

1. Before a Product can be inserted, the Store that the product will be listed in must already exist. This is because we need to include the StoreID (as a FK) of the store that the Product will be listed in when we insert the new Product.
2. Being a Primary Key, the value for ProductID must be unique.

The above adds a new Product to the database that is listed in the Store with StoreID=119.

To add a new Order to the database:
Adding a new order is as simple as inserting a new row into the AnOrder table:

INSERT INTO AnOrder (UserID, ProductID, OrderID, Email, AmountPaid, Date, PaymentType) VALUES ('2','3','1','glahlkga@algjakl.com','67.5','2/17/20','Venmo');

There are some requirements when doing this:

1. Before an order can be created, there must be an existing Product with the ProductID provided and there must be an existing User with the UserID provided. This is because both ProductID and UserID are foreign keys for the product.
2. Being a primary key, the value for OrderID must be unique.

The above creates an order placed by User with UserID=2 who ordered the Product with ProductID=3.

D. DELETE examples

There are likely 5 entities that you may want to delete records from in the database: **users, buyers**, **sellers**, **products**, and **orders.** We discuss each of them in turn and provide the relevant SQL code.

To delete a User from the database:
Say that we want to delete the user with the UserID of 2. Then we would do that with the following query:

DELETE FROM USER WHERE UserID = '2';

No other operations need to be performed in order to complete this. This is because for all other entities that depend upon UserID as a foreign key, such as BUYER and SELLER, the operation SET NULL is programmed to execute automatically upon the User's deletion. Then for these entities, upon this update that makes that PK null, the CASCADE function is programmed to execute automatically for any *further* entities that had a FK pointing to that PK. Thus those FK will become null as well. Thus, nothing but this query must be run in order to delete a user, and FKs that pointed either directly or transitively to the user will become null.

To delete a Buyer from the database:
Say that we want to delete the buyer with the BuyerId of 7. Then we would do that with the following query:

DELETE FROM BUYER WHERE BuyerID = '7';

No other operations need to be performed in order to complete this. This is because for all other entities that depend upon BuyerID as a foreign key, the operation SET NULL is programmed to execute automatically upon the Buyer's deletion.

To delete a Seller from the database:
Say that we want to delete the seller with the SellerID of 4. Then we would do that with the following query:

DELETE FROM SELLER WHERE SellerID = '4';

No other operations need to be performed in order to complete this. This is because for all other entities that depend upon SellerID as a foreign key, the operation SET NULL is programmed to execute automatically upon the Seller's deletion.

To delete a Product from the database:
Say that we want to delete the product with the ProductID of 34. Then we would do that with the following query:

DELETE FROM PRODUCT WHERE ProductID = '34';

No other operations need to be performed in order to complete this. This is because for all other entities that depend upon ProductID as a foreign key, the operation SET NULL is programmed to execute automatically upon the Product's deletion.

To delete an Order from the database:
Say that we want to delete the order with the OrderID of 75. Then we would do that with the following query:

DELETE FROM ORDER WHERE OrderID = '75';

No other operations need to be performed in order to complete this as no other entities rely upon OrderID as a foreign key.

Section 3 – Graded Checkpoint Documents

CP01

CSE 3241 Project Checkpoint 01
Team 6: Zhuo Xu

1. List names of all your team members. Provide a paragraph explaining how you have been working as a team under remote setup so far, how you plan to communicate with each other, share work, etc. Any issues related to time differences, technology constraints, etc?

   We have been using GroupMe to chat and plan meetings; Zoom for the meetings; and Google Drive for collaboration. Going forward we will continue doing this to communicate and share work.
   No time difference or technology constraint issues.

2. Based on the requirements given in the project overview, list the entities to be modeled in this database. For each entity, provide a list of associated attributes. Make sure that your design allows for proper handling of buyer/seller interactions such as orders, payments, feedback, and karma points.

Entity	Attributes for that entity
Users	User_ID, Name, Email, Password, Payment Info (a composite attribute, made up of Paypal Info, Karma Points, Venmo Info, Credit/Debit card info, and Cryptocurrency info), # of messages (derived)
Buyers (subclass of Users)	Inherits all the attributes of Users superclass
Sellers (subclass of Users)	Inherits all the attributes of Users superclass Adds the following attributes: Number_of_stores (derived), Bio, Photo
Stores	Store_ID, Name, Description, Banner, URLs, Accepted payment types
Products	Product_ID, Price, Title, Description, Availability (a boolean), File type, Download link, Size (say in bytes), Images, Keywords
Orders	Order_ID, Delivery Email, Amount paid, Date, Payment Type Used
Seller Reviews	Seller_Review_ID, Review
Product Reviews	Product_Review_ID, Review

3. Based on the requirements given in the project overview, what are the various relationships between entities? (For example, “CUSTOMER entities purchase IP Item entities”).

Each SELLER owns 0 to many STORE
Each STORE must be owned by 1 and only 1 SELLER
Each PRODUCT is listed in 1 and only 1 STORE
Each STORE can sell 0 to many PRODUCT
Each ORDER is created by 1 and only 1 BUYER
Each BUYER can make 0 to many ORDER
Each ORDER contains 1 and only 1 PRODUCT
Each PRODUCT can be in 0 to many ORDER
Each SELLER REVIEW is created by 1 and only 1 BUYER
Each SELLER REVIEW is connected to 1 and only 1 SELLER
Each BUYER can leave 0 to many SELLER REVIEW
Each SELLER can have 0 to many SELLER REVIEW
Each PRODUCT REVIEW is created by 1 and only 1 BUYER
Each BUYER can leave 0 to many PRODUCT REVIEW
Each PRODUCT REVIEW is connected to 1 and only 1 PRODUCT
Each PRODUCT can have 0 to many PRODUCT REVIEW

4. Propose at least two additional entities that it would be useful for this database to model beyond the scope of the project requirements. Provide a list of possible attributes for the additional entities and possible relationships they may have with each other and the rest of the entities in the database. Give a brief, one sentence rationale for why adding these entities would be interesting/useful to the stakeholders for this database project.

Entity	Attributes	Rationale
Shopping Cart (weak entity)	Number of Items (derived), Total Price of Cart (derived)	People may want to buy multiple products at a time, but a group of items from differents store, or save an item without buying it
Message	Message_ID, Subject, text, attachment, read	Message contains all of the relevant information related to a message -- buyers and sellers may want to communicate with each other

Each SHOPPING CART contains 0 to many PRODUCT
Each PRODUCT can be in 0 to many SHOPPING CART
Each BUYER has one and only one SHOPPING CART

Each USER sends 0 to many MESSAGE
Each USER receives 0 to many MESSAGE
Each MESSAGE is sent by only 1 USER
Each MESSAGE is received by only 1 USER

Note: when a buyer actually checks out a shopping cart -- i.e., when they actually pay for all the products in the cart -- that will create an Order for each purchased product. That is, if the shopping cart had 3 products inside, 3 order entities would be created.

5. Give at least four examples of some informal queries/reports that it might be useful for this database might be used to generate. Include one example for each of the additional entities you proposed in question 3 above.

   If a seller wants to generate a sales report of all the items they've sold, we can query the Orders table like "Give me all the rows where the seller in the order was this seller".

   If a seller wants to see all the items they are currently selling in store S, we can query the Products table like "Give me all the rows where Store_ID = <id of the Store> AND Availability = True"

   If a customer/buyer wants to access their shopping cart a query would be “Retrieve the shopping cart entity where the owner of the shopping cart is this buyer”

   If a user wants to see the number of messages they have, a query would be “Derive and return the attribute Number of messages for the user with user id=<my user id>”

   If a user wants to see all the messages to them, a query could be like "Show me all the message entities where this user was the recipient"

6. Suppose we want to add a new IP Item to the database. How would we do that given the entities and relationships you’ve outlined above? Is it possible to add up to five images for the IP Item? Is it possible for the IP Item to be purchase by more than one Payment Type? Is it possible for the Buyer to purchase IP Items from multiple Sellers at one time? Can a Buyer leave feedback on multiple items in the Seller’s store? Explain how your model supports these possibilities. If it does not, make changes that allow your design to support all these requirements.

   To add a new IP item, the seller would create a new Product entity that is linked to a Store entity, which is linked to the seller’s User entity.

   To add up to five images for the IP item, the user would add links to the five images under the Images multi-valued attribute for the Product entity.

   Each product would list the accepted payment types of the seller under Store and allow buyers to purchase using any of the accepted types; the actual type used by the buyer would be recorded within an Order entity.

   Buyers can purchase IP Items from multiple sellers at one time, Shopping Cart entity will support this by allowing multiple Product entities to be selected by buyers before purchasing (shopping cart creates orders for all products within it)

   Buyers can leave multiple feedback by creating multiple Product Review entities each linked to the Product entities that they are leaving feedback for.

7. Determine at least three other informal update operations and describe what entities would need to have attributes altered and how they would need to be changed given your above descriptions. Include one example for each of the additional entities you proposed in question 4 above.
    a) Change user email (the user’s email attribute will need to be changed)
    b) Change delivery email address for an Order (would need to update the delivery email address attribute for an order)
    c) Update the cart to contain a new item (add a new product entity inside the shopping cart. Nothing about the Product would change; the shopping cart's Number of Items and Total Price derived attributes would be updated though)

    d) Update a message once it has been read by the recipient (the "read" attribute of the message would be flipped to true, and defaults to false)

8. Provide an ER diagram for your database. Make sure you include all of the entities and relationships you determined in the questions above INCLUDING the entities for question 4 above, and remember that EVERY entity in your model needs to connect to another entity in the model via some kind of relationship. You can use draw.io for your diagram. If drawing on paper, make sure that your drawing is clear and neat. Ensure that you use a proper notation and include a legend.

Please see the attached .html file for our EERD. We found this was the nicest way to export it from draw.io (i.e., the result had the best quality this way). Just double click the .html file to open it in your browser, and there is our EERD (and you can zoom too).

NOTE: ‘Informal’ means stated in plain English, not in SQL or Relational Algebra.

Feedback on CP01

The Feedback	How it was addressed
Heres a couple questions to ask yourself for your diagram: How will this DB handle Payment type? (buyers can purchase with karma points, CC, or Cryptocurrency) Multiple credit cards or payment types per customer? Multiple images per IP Item and multiple images per Seller account? Multiple items per purchase? From different Sellers? If B&B changes or adds an IP Type, how many entities in DB have to be changed? From Logan Laurer	We addressed the feedback by answering each question, and making any needed changes. "How will this DB handle Payment type? (buyers can purchase with karma points, CC, or Cryptocurrency)" → **Good**. Each user a variety of payment info they can use. "Multiple credit cards or payments types per customer/buyer" → **Fixed**. Originally our database only supported one credit card per user. We fixed this by creating the HAS relationship in our EERD, so that a single user can have many credit cards. This relationship is visible in our EERD above; and we also added a Credit Card table (visible in our relational schema above) to account for this 1-to-many relationship. "Multiple images per IP Item" → **Good**. Our Images attribute was already multi-valued in the EERD; and this is handled in the Relational Schema using the 1-to-many relationship between Product and Product Image table. "Multiple images per Seller account" → **Fixed**. Originally we only allowed one image per seller account, so photo Attribute of the Seller entity was single-valued in our EERD. It is fixed to be multi-valued in the EERD now; and corresponding table (Seller Photos) is used to facilitate that 1-to-many relationship. "Multiple Items per purchase? From Different Sellers?" → **Good.** It is true that an Order can only contain one Product in our system. How this is handled though is that the Shopping Cart can contain many products, and these products can be from different Sellers too. Then, when the user hits "checkout", what would happen is that one order would be created for every Product in their shopping cart. **In this way, users can purchase multiple IP items (which are perhaps from different sellers) all at the same time**.

	"If B&B changes or adds an IP Type, how many entities in DB have to be changed?" --> **Good**. Adding a new Product with a different file type is perfectly possible in the system.

CP02

CSE 3241 Project Checkpoint 02 – Relational Model and Relational Algebra
Team 6: Zhuo Xu

In a **NEATLY TYPED** document, provide the following:

1. **Provide a current version of your ER Model as per Project Checkpoint 01. If you were instructed to change the model for Project Checkpoint 01, make sure you use the revised version of your ER Model.**

   Please see the attached HTML file called "Project EERD v2".
   This is our current EERD. We did not have to make any changes to it; the only reason we made a second version is to make the EERD look nicer. But all the entities, attributes, relationships etc are still the exact same -- version 2 just looks nicer than the original.

2. Map your ER model to a relational schema. Indicate all primary and foreign keys.

Primary keys are underlined and foreign keys are bolded and described beneath the relation.

USER

<u>User ID</u>	Name	Email	Password	Paypal Info	Venmo Info	Karma Points	Credit/Debit	Cryptocurrency info

CREDIT CARD

User_ID	<u>Card_number</u>	CVV	Cardholder_Name	Exp_Date

User_ID is foreign key from USER(user id)

BUYER

Buyer_ID

Buyer_ID is foreign key from USER(user id)

SELLER

Seller_ID	Bio

Seller_ID is foreign key from USER(user id)

SELLER_PHOTOS

Seller_Photo_ID	**Seller_ID**	Photo

Seller_ID is foreign key pointing to Seller_ID

STORE

Store_ID	Name	Store_Description	Banner	URLs	**Seller_User_ID** (the id of the seller that owns the store)

Seller_User_ ID is a foreign key from USER

STORE ACCEPTED PAYMENT TYPES

Store_ID	Accepted Payment

Store_ID is a foreign key from STORE

STORE URLS

Store_ID	URL

Store_ID is a foreign key from STORE (i.e., pointing back to tuple/row in the STORE relation/table)

PRODUCT

Store ID	Product_ID	Price	Title	Descrip-tion	Availability	File Type	Download Link	Size	Images	Keywords

Store ID is a foreign key from STORE

PRODUCT_IMAGES

Product_ID	Image

Product_ID is a foreign key from PRODUCT

PRODUCT_KEYWORDS

Product_ID	Keyword

Product_ID is a foreign key from PRODUCT

PRODUCT REVIEW

Product ID	**User_ID**	Product_Reviews_ID	Review

Product ID is a foreign key from PRODUCT
User ID is a foreign key from BUYER

ORDER

User ID	Order _ID	Delivery Email	Amount paid	Date	Payment Type Used	**Product_ID**

User ID is foreign key from BUYER
Product ID is foreign key from PRODUCT (i.e., this points back to the 1 product for this order)

MESSAGE

Sender User ID	**Receiver User ID**	Message_ID	Subject	Text	Attachment	Read	Sent Time

Sender User ID is a foreign key from USER
Receiver User ID is a foregin key from USER

SHOPPING CART

Shopping_Cart_ID

Shopping_Cart_ID is a foreign key equal to User_ID from USERS (this is the buyer for this particular shopping cart)

SHOPPING CART PRODUCT

Shopping_Cart_ ID	**Product ID**

Shopping Cart ID is a foreign key from SHOPPING CART
Product ID is a foreign key from PRODUCT

SELLER REVIEW

Seller_Revew_ID	Review	**Buyer_User_ID** (the person doing the reviewing)	**Seller_User_ID** (the person being reviewed)

Buyer_User_ID and **Seller_User_ID** are foreign keys from USER table/relation

3. Given your relational schema, provide the relational algebra to perform the following queries. If your schema cannot provide answers to these queries, revise your ER Model and your relational schema to contain the appropriate information for these queries:

a. Find the titles of all IP Items by a given Seller that cost less than $10 (you choose how to designate the seller)

Let the seller that we want to find have Seller_ID of Seller_ID_To_Select

Titles $\leftarrow \mathbf{\Pi}_{\text{Title}} (\sigma_{\text{Price} < 10}(\sigma_{\text{Seller\_ID = Seller\_ID\_To\_Select}}((\text{SELLER} \bowtie_{\text{Seller\_ID = Seller\_ID}} \text{STORE}) \bowtie_{\text{Store\_ID = Store\_ID}} \text{PRODUCT})))$

b. Give all the titles and their dates of purchase made by given buyer (you choose how to designate the buyer)

Let the buyer that we want to have Buyer_ID of Buyer_ID_To_Select

OrderInfo $\leftarrow \mathbf{\Pi}_{\text{Title, Date}}(\sigma_{\text{Buyer\_ID = Buyer\_ID\_To\_Select}} ((\text{BUYER} \bowtie_{\text{Buyer\_ID = User\_ID}} \text{ORDER}) \bowtie_{\text{Product\_ID = Product\_ID}} \text{PRODUCT}))$

c. Find the seller names for all sellers with less than 5 IP Items for sale

Names $\leftarrow \mathbf{\Pi}_{\text{Name}} (\text{USER} \bowtie_{\text{User\_ID = Seller\_ID}} (\sigma_{\text{COUNT Product\_ID} < 5} ({}_{\text{Seller\_ID}} F_{\text{COUNT Product\_ID}} ((\text{SELLER} \bowtie_{\text{Seller\_ID = Seller\_ID}} \text{STORE}) \bowtie_{\text{Store\_ID = Store\_ID}} \text{PRODUCT}))))$

d. Give all the buyers who purchased a IP Item by a given seller and the names of the IP Items they purchased

Let the seller that we want to find have Seller_ID of Seller_ID_To_Select

BuyersAndItemNames $\leftarrow \mathbf{\Pi}_{\text{Buyer ID, Product Title}}(\text{BUYER} \bowtie_{\text{Buyer\_ID=Buyer\_ID}} (\text{ORDER} \bowtie_{\text{Product\_ID = Product\_ID}} (\sigma_{\text{Seller\_ID = Seller\_ID\_To\_Select}} ((\text{SELLER} \bowtie_{\text{Seller\_ID = Seller\_ID}} \text{STORE}) \bowtie_{\text{Store\_ID=Store\_ID}} \text{PRODUCT})))$

e. Find the total number of IP Items purchased by a single buyer (you choose how to designate the buyer)

Assuming that the we want to get the total number of IP Items purchased by a buyer with ID "Buyer_ID_To_Select" …

BuyerOrders $\leftarrow$ Order $\bowtie_{\text{User\_ID=Buyer\_ID}}$ Buyer
OrdersForOurSpecificBuyer $\leftarrow \sigma_{\text{Buyer\_ID=Buyer\_ID\_To\_Select}}$ BuyerOrders
Result $\leftarrow F_{\text{COUNT Order\_ID}}$(OrdersForOurSpecificBuyer)

Result should be a relation like this:

Count
23

And 23 would be the number of IP items purchased by our buyer.

f. Find the buyer who has purchased the most IP Items and the total number of IP Items they have purchased

BuyerOrders ⟵ Order $\bowtie_{User\_ID=Buyer\_ID}$ Buyer
OrderCountsForEachBuyer ⟵ $_{Buyer\_ID}F_{COUNT\ Order\_ID}$(BuyerOrders)

MaxNumberOfItemsPurchased ⟵ $F_{MAX\ COUNT\_Order\_ID}$(OrderCountsForEachBuyer)

BuyerWithMaxItemsPurchased ⟵ $\sigma_{COUNT\_Order\_ID\ =\ MaxNumberOfItemsPurchased.MAX}$ (OrdersCountsForEachBuyer)

BuyerWithMaxItemsPurchased relation should look like this:

Buyer_ID	COUNT_Order_ID
12	23

Which gives the id of the buyer with most items purchased, and the amount of items they have purchased in the Count_Order_IDs columns

If we want some actual info on this buyer, we can join it with the USER table to get their name and such:
DesiredBuyer ⟵ USER $\bowtie_{User\_ID\ =\ Buyer\_ID}$ BuyerWithMaxItemsPurchased

4. Three additional interesting queries in plain English and also relational algebra. Your queries should include at least one of these:
a. outer joins
b. aggregate function
c. “extra” entities from CP01

Find the price of items in a specific users shopping cart where the specific user’s id will be denoted by Desired_User_ID
DesiredUserCart ⟵ USER $\bowtie_{User\_ID\ =\ Desired\_User\_ID}$ SHOPPING CART
ItemsInCart ⟵ (DesiredShoppingCart $\bowtie_{Shopping\_Cart\_ID\ =\ Shopping\_Cart\_ID}$ SHOPPING CART PRODUCT) $\bowtie_{Product\_ID\ =\ Product\_ID}$ PRODUCT
CartPrice ⟵ $F_{SUM\ Price}$ (ItemsInCart)

Find the number of Items in a specific users shopping cart where the specific user’s id will be

denoted by Desired_User_ID
DesiredUserCart ⟵ USER $\bowtie_{\text{User\_ID = Desired\_User\_ID}}$ SHOPPING CART
ItemsInCart ⟵ (DesiredShoppingCart $\bowtie_{\text{Shopping\_Cart\_ID = Shopping\_Cart\_ID}}$ SHOPPING CART PRODUCT) $\bowtie_{\text{Product\_ID = Product\_ID}}$ PRODUCT
CartSize ⟵ F $_{\text{COUNT Product\_ID}}$ (ItemsInCart)

Find the UserId and StoreId for all users. Include users who don't have a store.

UserStore ⟵ $\Pi_{\text{User\_Id, Seller\_user\_id}}$ (USER ⟕ $_{\text{User\_Id = Seller\_user\_id}}$ STORE)

Feedback on CP02

The Feedback	How it was addressed
I don't see any personally, but be sure not to have any FKs or surrogate keys in ERD. Be sure your db can handle: - Buyers purchasing with karma points, CC, or Cryptocurrency) -Multiple credit cards/payment types per buyer -Multiple images per IP Item/multiple images per Seller account? -Multiple items per purchase and from different Sellers -If B&B changes or adds an IP Type, how many entities in DB have to be changed? i recommend putting this into table form on another document, then creating arrows to indicate primary/foreign keys with other entities From Logan Laurer	-We decided that users could have multiple credit cards and addressed this issue in our ER Diagram and schema. However, we are enforcing that Users will have only one Venmo, Paypal, or Cryptocurrency account on file at a given time. -We now allow multiple images for products and sellers as seen in ER diagram and schema. -We decided that when a user checks out, every product purchased at that time will have its own order created in the table. We also did end up putting the relational scheme into table form, to make it prettier and easier to read. You can see the finished product in Section 1, Part B, "Relational Schema / Database Schema" above.

CP03

CSE 3241 Project Checkpoint 03 – SQL and More SQL
Team 6: Zhuo Xu

You will be submitting several nicely formatted files for this checkpoint. Provide the following:

1. Provide a current version of your ER Diagram and Relational Model as per Project Checkpoint 02. **If you were instructed to change the model for Project Checkpoint 02, make sure you use the revised versions of your models**

Current EERD is here:
https://app.diagrams.net/#G1QDUK0f5w8Tw6SQGziGqVu2e95kBzTR4B

And current Relational Model / Database Schema is here:
https://app.diagrams.net/#G1m5BjahX31KLEb7JLvtUUgCx0AKgxpdqv

2. Given your relational schema, create a text file containing the SQL code to create your database schema. Use this SQL to create a database in SQLite. Populate this database with the data provided for the project as well as 20 sample records for each table that does not contain data provided in the original project documents.

txt file created for submission

3. Given your relational schema, provide the SQL to perform the following queries. If your schema cannot provide answers to these queries, revise your ER Model and your relational schema to contain the appropriate information for these queries. These queries should be provided in a plain text file named "SimpleQueries.txt":

txt file created for submission

a. Find the titles of all IP Items by a given Seller that cost less than $10 (you choose how to designate the seller)

   Assume we designate the Seller by Name.
   Here is the query with Seller Name = 'fajf jaifja'

   SELECT Product.Title FROM Product, Store, Seller, User WHERE Product.Price<10 AND Product.StoreID=Store.StoreID AND Store.SellerID = Seller.SellerID AND Seller.SellerID=User.UserID AND User.Name='fajf jaifja';

b. Give all the titles and their dates of purchase made by given buyer (you choose how to designate the buyer)

   Assume that by "Title of Purchase" we mean the name of the item purchased,
   And assume that we specify a Buyer by Name.
   Here is the query with Buyer Name = 'fajf jaifja'

   SELECT Product.Title AS ProductPurchased, AnOrder.Date FROM AnOrder, Product, User WHERE AnOrder.ProductID=Product.ProductID AND AnOrder.UserID=User.UserID AND User.Name='fajf jaifja';

c. Find the seller names for all sellers with less than 5 IP Items for sale

   SELECT User.Name FROM Product, Store, Seller, User WHERE Product.Available='TRUE' AND Product.StoreID=Store.StoreID AND Store.SellerID=Seller.SellerID AND Seller.SellerID=User.UserID GROUP BY User.Name HAVING Count(User.Name) < 5;

d. Give all the buyers who purchased a IP Item by a given seller and the names of the IP Items they purchased

   Here is the query with Seller Name = 'falf iejfla'

   SELECT BuyerUser.Name AS NameOfBuyer, Product.title AS ProductPurchased FROM AnOrder, Product, Store, Seller, User AS BuyerUser, User AS SellerUser WHERE AnOrder.UserID=BuyerUser.UserID AND AnOrder.ProductID=Product.ProductID AND Product.StoreID=Store.StoreID AND Store.SellerID=Seller.SellerID AND Seller.SellerID=SellerUser.UserID AND SellerUser.Name='falf iejfla';

e. Find the total number of IP Items purchased by a single buyer (you choose how to designate the buyer)

   Assume we specify the Buyer by Name.
   Here is the query with Buyer Name = 'fajefai jfiajfl'

   SELECT Count(AnOrder.UserId) AS NumItemsPurchased FROM AnOrder, User WHERE AnOrder.UserID=User.UserID AND User.Name='fajefai jfiajfl';

f. Find the buyer who has purchased the most IP Items and the total number of IP Items they have purchased

```
SELECT BuyerName, MAX(CT) AS NumItemsPurchased FROM
        (SELECT BuyerUser.Name AS BuyerName, Count(AnOrder.UserID) as CT
        FROM AnOrder, User as BuyerUser
        WHERE AnOrder.UserID = BuyerUser.UserID
        GROUP BY AnOrder.UserID);
```

4. For Project Checkpoint 02, you were asked to come up with three additional interesting queries that your database can provide. Provide the SQL to perform those queries. Your queries should include at least one of these:

a. outer joins
b. aggregate function (min, max, average, etc)
c. “extra” entities from CP01

Find the UserId and StoreId for all users. Include users who don’t have a store:

```
SELECT U.UserID, S.StoreID
FROM User AS U LEFT OUTER JOIN Store AS S ON U.UserID=S.SellerID;
```

Find the price of items in a specific users shopping cart where the specific user’s id will be denoted by Desired_User_ID:

```
SELECT sum(P.price)
FROM User AS U, ShoppingCart AS SC, ShoppingCartProduct AS SCP, Product AS P
WHERE U.UserID=‘Desired_User_ID’ AND U.UserID=SC.ShoppingCartID AND
SC.ShoppingCartID=SCP.ShoppingCartID AND SCP.ProductID=P.ProductID;
```

Find the number of Items in a specific users shopping cart where the specific user’s id will be denoted by Desired_User_ID:

```
SELECT count(*)
FROM User AS U, ShoppingCart AS SC, ShoppingCartProduct AS SCP, Product AS P
WHERE U.UserID=‘Desired_User_ID’ AND U.UserID=SC.ShoppingCartID AND
SC.ShoppingCartID=SCP.ShoppingCartID AND SCP.ProductID=P.ProductID;
```

If your schema cannot provide answers to these queries, revise your ER Model and your relational schema to contain the appropriate information for these queries. These queries should be provided in a plain text file named “ExtraQueries.txt”

5. Given your relational schema, provide the SQL for the following more advanced queries. These queries may require you to use techniques such as nesting, aggregation using having clauses, and other SQL techniques.

If your database schema does not contain the information to answer to these queries, revise your ER Model and your relational schema to contain the appropriate information for these queries.

Note that if your database does contain the information but in non-aggregated form, you should NOT revise your model but instead figure out how to aggregate it for the query!

These queries should be provided in a plain text file named “AdvancedQueries.txt”

a. Provide a list of buyer names, along with the total dollar amount each buyer has spent.

```
SELECT Name, sum(AmountPaid) AS TotalSpent
FROM Buyer AS B, User AS U, AnOrder AS O
WHERE BuyerID=U.UserID AND O.UserID=BuyerID
GROUP BY BuyerID;
```

b. Provide a list of buyer names and e-mail addresses for buyers who have spent more than the average buyer.

```
SELECT Name, U.Email
FROM Buyer AS B, User AS U, AnOrder AS O
WHERE BuyerID=U.UserID AND O.UserID=BuyerID
GROUP BY BuyerID
HAVING sum(amountPaid) >
        (SELECT avg(TotalSpent) FROM (
                SELECT Name, sum(AmountPaid) AS TotalSpent
                FROM Buyer AS B, User AS U, AnOrder AS O
                WHERE BuyerID=U.UserID AND O.UserID=BuyerID
                GROUP BY BuyerID));
```

c. Provide a list of the IP Item names and associated total copies sold to all buyers, sorted from the IP Item that has sold the most individual copies to the IP Item that has sold the least.

```
SELECT ProductID, count(*)
FROM AnOrder
GROUP BY ProductID
ORDER BY count(*) DESC;
```

d. Provide a list of the IP Item names and associated dollar totals for copies sold to all buyers, sorted from the IP Item that has sold the highest dollar amount to the IP Item that has sold the smallest.

```
SELECT Product.ProductID, Title, SUM(Product.Price) as TotalDollarProfitsFromThisProduct
```

FROM Product, AnOrder, User
WHERE AnOrder.ProductId=Product.ProductId AND AnOrder.UserID=User.UserID
GROUP BY Product.ProductID
ORDER BY TotalDollarProfitsFromThisProduct DESC;

e. Find the most popular Seller (i.e. the one who has sold the most IP Items)

CREATE VIEW Temp (SellerID, ProductNum) AS SELECT Seller.SellerID, count(Seller.SellerID) FROM ((Seller JOIN Store ON Seller.SellerID = Store.SellerID) JOIN Product ON Store.StoreID = Product.StoreID) GROUP BY Seller.SellerID;
SELECT SellerID FROM Temp WHERE ProductNum = (SELECT max(ProductNum) from TEMP);

f. Find the most profitable seller (i.e. the one who has brought in the most money)

```
 Select MAX(TotalMadeByThisSeller), IdOfSeller, NameOfSeller
FROM
        (SELECT SUM(Product.Price) As TotalMadeByThisSeller,  User.UserID as
        IdOfSeller, User.Name AS NameOfSeller
        FROM AnOrder, Product, Store, Seller, User
         WHERE AnOrder.ProductId=Product.ProductID AND
        Product.StoreID=Store.StoreId AND Store.SellerId=Seller.SellerId AND
        Seller.SellerID=User.UserID
        GROUP BY NameOfSeller);
```

g. Provide a list of buyer names for buyers who purchased anything listed by the most profitable Seller.

```
SELECT DISTINCT(U.Name)
FROM User AS U, Buyer AS B, AnOrder AS O, Product AS P, Store AS S, Seller
WHERE B.BuyerID=U.UserID AND O.UserID=B.BuyerID AND O.ProductID=P.ProductID
AND P.StoreID=S.StoreID AND Seller.SellerID=
(SELECT IdOfSeller
FROM
        (Select MAX(TotalMadeByThisSeller), IdOfSeller, NameOfSeller
        FROM
                (SELECT SUM(Product.Price) As TotalMadeByThisSeller,  User.UserID
                as IdOfSeller, User.Name AS NameOfSeller
                FROM AnOrder, Product, Store, Seller, User
```

WHERE AnOrder.ProductId=Product.ProductID AND
Product.StoreID=Store.StoreId AND Store.SellerId=Seller.SellerId AND
Seller.SellerID=User.UserID
GROUP BY NameOfSeller)));

h. Provide the list of sellers who listed the IP Items purchased by the buyers who have spent more than the average buyer.

CREATE VIEW AmountSpent (BuyerID, Amount) AS SELECT BuyerID, sum(AmountPaid) FROM (Buyer JOIN AnOrder ON BuyerID = UserID) GROUP BY BuyerID;
CREATE VIEW BigBuyers (BuyerID) AS SELECT BuyerID FROM AmountSpent WHERE Amount > (SELECT avg(Amount) FROM AmountSpent);
CREATE VIEW Purchases (ProductID) AS SELECT distinct Product.ProductID FROM ((BigBuyers JOIN AnOrder ON BigBuyers.BuyerID = AnOrder.UserID) JOIN Product ON AnOrder.ProductID = Product.ProductID);
SELECT distinct Seller.SellerID FROM (((Purchases JOIN Product ON Purchases.ProductID = Product.ProductID) JOIN Store ON Product.StoreID = Store.StoreID) JOIN Seller ON Store.SellerID = Seller.SellerID);

6. Once you have completed all of the questions for Part Two, create a ZIP archive containing the binary SQLite file and the three text files and submit this to the Carmen Dropbox. Make sure your queries work against your database and provide your expected output before you submit them!

Feedback on CP03

The Feedback	How it was addressed
diagram and queries look good - I do not have any additional suggestions From Logan Laurer	No corrections needed.

CP04

CSE 3241 Project Checkpoint 04 - Functional Dependencies, Normal Forms, Indexes, Transactions
Team 6: Zhuo Xu

In a **NEATLY TYPED** document, provide the following:

1. **Provide a current version of your ER Diagram and Relational Model as per Project Checkpoint 03. If you were instructed to change the model for Project Checkpoint 03, make sure you use the revised versions of your models.**

For ER Diagram, see file "Current EERD.html"
For Database Schema, see file "Current Database Schema.html"

2. **For each relation schema in your model, indicate the functional dependencies. Think carefully about what you are modeling here - make sure you consider all the possible dependencies in each relation and not just the ones from your primary keys. For example, a customer's credit card number is unique, and so will uniquely identify a customer even if you have another key in the same table (in fact, if the customer can have multiple credit card numbers, the dependencies can get even more involved).**

User

- {UserID} → {Name, Email, Password, Paypal Info, Venmo Info, Karma Points, Cryptocurrency Info}
- Email -> UserID
- Email -> Name
- Email -> Password
- Email -> Paypal Info
- Email -> Venmo Info
- Email -> Karma Points
- Email -> Cryptocurrency Info
- Note that an Email always determines a user; we are assuming that one email cannot be associated with multiple users
- Table in **2NF.** All values are atomic (so 1NF); all non-key attributes are full dependent on the key (so 2NF); but there are some transitive dependencies (eg, UserID key -> Email, and Email -> Name)
- **Justification for not bringing this table to 3NF**: The idea with normalization is that it forces you to remove duplicate data in your tables. Normally for tables in 2NF but not 3NF, going to 3NF will allow you to remove data redundancy (i.e., have data defined in only one spot). However, in this case there is a 1-to-1 mapping between UserIDs and Emails (i.e., one UserID is always exactly associated with one Email; and one Email is always exactly associated with one user). Hence, there would be no duplication of data

here. So we argue that it would make the design more complicated to split the table up and achieve 3NF. Hence we leave the table as is.

Buyer

- BCNF

Credit/Debit Card

- {CardNumber} ⟶ {CVV, Exp Date, Cardholder Name, UserID}
- BCNF

Shopping Cart

- BCNF

Shopping Cart Product

- BCNF

Seller

- {SellerID} ⟶ {Bio}
- BCNF

Seller Photos

- {SellerPhotoID} ⟶ {SellerId, Photo}
- BCNF

Store

- {StoreID} ⟶ {SellerID, Name, Description, Banner}
- BCNF

Store Accepted Payment Types

- BCNF

Seller Review

- {ReviewID} ⟶ {BuyerID, SellerID, Review}
- BCNF

Store URL

- BCNF

Product

- {ProductID} → {StoreID, Price, Title, Description, Availability, File Type, Download Link, Size}
- BCNF

Product Image

- {ProductImageID} → {ProductID, Image}
- BCNF

Product Keyword

- {ProductKeywordID} → {ProductID, Keyword}
- BCNF

Product Review

- {ReviewID} → {ProductID, UserID, Review}
- BCNF

Order

- {OrderID} → {UserID, ProductID, DeliveryEmail, AmountPaid, Date, PaymentType}
- BCNF

Message

- {MessageID} → {Sender, Receiver, Subject, Text, Attachment, Read, SentTime}
- BCNF

3**. For each relation schema in your model, determine the highest normal form of the relation. If the relation is not in 3NF, rewrite your relation schema so that it is in at least 3NF.**

All tables in 3NF besides USERS. Justification for leaving in 2NF above.

4. For each relation schema in your model that is in 3NF but not in BCNF, either rewrite the relation schema to BCNF or provide a short justification for why this relation should be an exception to the rule of putting relations into BCNF.

Every relation above is in BCNF besides USERS. Justification is above.

5. For your database, propose at least two interesting views that can be built from your relations. These views must involve joining at least two tables together each and must include some kind of

aggregation in the view. Each view must also be able to be described by a one or two sentence description in plain English. Provide the code for constructing your views along with the English language description of what the view is supposed to be providing.

View 1: Show all the Products for all the Stores, and the owner of each Store

```
CREATE VIEW ProductsInStores
AS   SELECT User.Name AS NameOfStoreOwner, Store.StoreID, Store.Name AS StoreName,
ProductID, Title AS ProductTitle
        FROM Store, Product, Seller, User
        WHERE Store.StoreID = Product.StoreID AND Store.SellerID=Seller.SellerID AND
Seller.SellerID=User.UserID
        ORDER BY NameOfStoreOwner
```

This view does as it says -- it shows all the products, for all the stores, and it shows the name of the owner for each store. Results are sorted by owner name.

View 2: Get number of products purchased from each store

```
CREATE VIEW ProductNum (UserID, NumProducts)
          AS SELECT S.StoreID, count(ProductID) FROM Store AS S, Product AS P, Order AS O
          WHERE S.StoreID = P.StoreID AND P.ProductID = O.ProductID
          GROUP BY S.StoreID
```

This query first joins Stores and Products based on StoreID and then joins this with Orders based ProductID. From this joined table, the number of products corresponding to each seller is calculated and returned as a view.

6. Description of two indexes that you want to implement in your DB. Explain their purpose and what you want to achieve by implementing them. Explain what type of indexing would be most appropriate for each one of them (Clustering, Hash, or B-tree) and why.

Index 1: Index on “ProductID” within the order table.

If we use **Hash-Table** indexing, then we could have very quick lookups of orders containing certain products, which is an action which could come in handy. To implement the hash table, the hash value would take a certain product ID to a certain value and all orders with a certain product id would end up in the same entry in the table. In fact, a valid hash function that would probably work well would be just returning the productId. Then each entry in the hash table would hold orders with the productIDs equal to that entry’s id.

Index 2: An index on the "price" column in the Products table

The idea here is to make lookups based on the Price of a product fast, since users would probably want to do that frequently.

Specifically, we would make this a **B-tree** index, because it's likely that people will lookup product based on price *ranges*, and B-tree are best for queries involving ranges.

For instance, an end-user may want to lookup all products where price < $10.00. If we had this B-tree index on the Price column, that would make this lookup faster. Hence, we think a B-tree index would be a good idea.

We figure it would be more common for an end-user to ask "give me all products where price < 10.00" (a range) then "give me all products where price exactly = 10.00" (an exact-value query). Hence, we would use a B-tree index over a Hash-table index here. Because B-tree indexes are best for range queries, whilst Hash-table indexes are best for exact-value queries.

7. Two sample transactions that you want to establish in your DB. Clearly document their purpose and function. Include the sample SQL code for each transaction. Each transaction should include read and/or write operations on at least two tables, with appropriate error and constraint checks and responses.

<u>**Transaction 1**</u>

```
BEGIN TRANSACTION GiveSellersKarmaPoints
        UPDATE USER SET Karma = Karma + 100 WHERE UserID IN (SELECT SellerID FROM SELLER);
        IF error THEN GO TO UNDO; END IF;
        COMMIT;
        GO TO FINISH;
        UNDO: ROLLBACK;
        FINISH:
END TRANSACTION;
```

This transaction adds 100 karma points to any user that is a seller. This could be useful if there was a time when the managers B&B wanted to give users who owned a store a discount. This would be done by giving sellers karma points for free. This transaction performs one read and one write operation. A read on Seller and a write on User.

<u>**Transaction 2**</u>

```
 BEGIN TRANSACTION GetAllShoppingCartItems
        SELECT ProductID FROM ShoppingCartProduct WHERE ShoppingCartID IN (SELECT BuyerID
                                                                            FROM Buyer);
        IF error THEN GO TO UNDO; END IF;
        COMMIT;
        GO TO FINISH;
        UNDO: ROLLBACK;
        FINISH:
END TRANSACTION;
```

This transaction retrieves all the items that are currently in a shopping cart. The purpose of this transaction is that it is efficient since it only looks at shopping carts for users that are currently buyers. It performs 2 read operations, one on ShoppingCartProduct and one on Buyer.

Feedback on CP04

The Feedback	How it was addressed
user should have total participation to buyer/seller no other comments - looks good From Logan Laurer	Our correction can be seen in our ER Diagram where there is now a double line from User to Buyer/Seller

Part 2 – The SQL Database

For our binary database, please see the **project.db** file in the "Binary Database" folder, which will be included with the submission.
This file is ready to be opened in SQLiteOnline.com.

For our SQL code, please see the following files in the "Text Files with SQL Code" folder:

- **CreateAndPopulateDB.txt** -- code to create the tables and populate them with sample data
- **CreateTwoViews.txt** -- the code to create the two views we discussed above
- **CreateTwoIndexes.txt** -- the code to create the two indexes we discussed above
- **SimpleQueries.txt** -- some basic SQL queries
- **ExtraQueries.txt** -- some more SQL queries
- **AdvancedQueries.txt** -- some advanced SQL queries
- **RunTwoTransactions.txt** -- the code to run the two transactions we discussed above

Thank you for reading!